HEAPS

Heaps in Theory

* uses a graphical tree to ***represent*** an unsorted array
* the tree
  + is a RBT (Regular Binary Tree)
    - so only 2 children
  + ***is completed from the top down, left to right (called complete)***
  + each node in the tree represented in the corresponding array
* cannot have duplicates
* items are added to the array in order (or make a COMPLETE tree)
* order of inputs does have an effect on the overall order of the heap

|  |  |
| --- | --- |
| Tree Representation | Array Representation |
| **[1]**  **[3]**  **[2]**  **[7]**  **[6]**  **[5]**  **[4]**  **[0]**  **[8]**  **[10]**  **[9]**  **// JUST A GRAPHICAL REPRESENTATION!!!** | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | | **-1** | **29** | **85** | **8** | **93** | **23** | **88** | **73** | **44** | **36** | |

// Draw the value in the elements of the tree from the array representation

Was does complete mean?

Minimum Binary Heap

* same constructions as a heap, but the minimum value of the ***entire tree*** is stored at the root
* the further down we go in the min heap, the value increases
  + parent will ALWAYS be ***less than or equal*** in value than the kids
  + this is called partial ordering

|  |
| --- |
| The Min Heap Structure |
|  |
| Notice 4 is the smallest value so far in this heap  Anything below the parent (no matter where) is >= than the parent  Notice the max value will be SOMEWHERE near the bottom |

**Initial class setup – BinaryHeap**

* code given uses an array/vector
  + default size is 10
  + calls buildHeap() just to do that

|  |
| --- |
| MinBH Construct(or) |
| Java |
| /\*\*  \* Construct the binary heap given an array of items.  \*/  **public** BinaryHeap( AnyType [ ] items )  {  currentSize = items.length;  array = (AnyType[]) **new** Comparable[ ( currentSize + 2 ) \* 11 / 10 ];  **int** i = 1;  **for**( AnyType item : items )  { array[ i++ ] = item; }  buildHeap( );  } |
| C++ |
| explicit BinaryHeap( const vector<Comparable> & items )  : array( items.size( ) + 10 ), currentSize( items.size( ) )  {  for( int i = 0; i < items.size( ); i++ )  array[ i + 1 ] = items[ i ];  buildHeap( );  } |

**Determining the relationships using code/array**

* Determining who is parent/child of a certain node is easy!!
  + using array notation and structure!!

|  |  |
| --- | --- |
| Tree Representation | |
| **[1]**  **[3]**  **[2]**  **[7]**  **[6]**  **[5]**  **[4]**  **[0]**  **[8]**  **[10]**  **[9]**  **// JUST A GRAPHICAL REPRESENTATION!!!** |

***Remember this is using an ARRAY representation!!!***

|  |  |  |
| --- | --- | --- |
| To Find | Formula | Answer |
| Parent index | **floor**((index)/2) | [6]/2 = 3  6’s Parent is 3 |
| Left Child index | 2(index) | 2\*[3] = 6  3’s Left Child is 6 |
| Right Child index | 2(index) + 1 | 2\*[3] + 1 = 7  3’s Right Child is 7 |
| **9’s Parent** | **what’s the formula?** | **using the formula, what’s the answer?** | |
| **2’s Left Child** |  |  | |
| **4’s Parent** |  |  | |

**Building and Inserting into a Min Heap**

* notice I do have to ***specify*** Minimum Binary Heap
* algorithm
  + place new node at END of array
    - next available complete spot in BT (top to bottom, left to right)
  + at end, could be in wrong order (parent is larger!)
    - continuously swap with parent going up the tree until parent < new node
    - this is called sift up or percolate up
  + notice that “lighter” values do bubble up, (maybe not to the root), but are in a higher position

|  |
| --- |
| **Inserting into an establish Min Heap** |
| Where will the next value be first placed, no matter the value?? |

|  |
| --- |
| Inserting a 6 |
|  |
| Start checking position!! (Check immediate Parent 17) |
|  |
| Swap since Parent is > new node!! |
|  |
| Start checking position!! (Check immediate Parent 17) |
|  |
| But 5 < 6, so we’re good!! |

First Example of Building a Heap: <http://youtu.be/m1Oa2Xk--3Y>

|  |
| --- |
| **Answer to in-class example** |
|  |

Try these on your own, insert in the order given, into a ***min*** heap:

1. 56, 43, 12, 67, 92, 4, 87, 53, 44, **93**
2. 61, 23, 57, 12, 68, 24, **14**, 96, 75, 63
3. 54, 76, 3, 83, 23, 45, 15, 35, 25, 14, 30
4. If I added 4 to #2, how many swaps would take place?

Answersb:

**Insert – the function**

* notice it checks the size of the array first
  + adds more if not enough
* ***temporarily*** spaces our value in [0]
* < 0 is not the value, but if there is a parent that is greater, then keep swapping

|  |
| --- |
| **Array version of Insert for MinBH** |
| Java |
| /\*\*  \* Insert into the priority queue, maintaining heap order.  \* Duplicates are allowed.  \* **@param** x the item to insert.  \*/  **public** **void** insert( AnyType x )  {  // check if size of array is enough to hold new node  **if**( currentSize == array.length - 1 )  { enlargeArray( array.length \* 2 + 1 ); }  // Percolate up  **int** hole = ++currentSize;  **for**( array[ 0 ] = x; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )  { array[ hole ] = array[ hole / 2 ]; }    // now put our new value into the right place  array[ hole ] = x;  }  Why is it /= 2? |
| C++ |
| void insert( const Comparable & x )  {  if( currentSize == array.size( ) - 1 )  array.resize( array.size( ) \* 2 );  // Percolate up  int hole = ++currentSize;  for( ; hole >= 1 && x < array[ hole / 2 ]; hole /= 2 )  array[ hole ] = array[ hole / 2 ]; // swap, from child to parent  array[ hole ] = x;  } |

Finding the minimum in a MinBH

* super easy!
* minimum value will ALWAYS be the root
  + if everything percolated correctly
  + [1] not [0]!!

|  |
| --- |
| **Min is always at the top of a MinBH** |
|  |

|  |
| --- |
| findMin the function |
| Java |
| /\*\*  \* Find the smallest item in the priority queue.  \* **@return** the smallest item, or throw an UnderflowException if empty.  \*/  **public** AnyType findMin( )  {  **if**( isEmpty( ) )  { **throw** **new** UnderflowException( ); }  **return** array[ 1 ];  } |
| C++ |
| const Comparable & findMin( ) const  {  if( isEmpty( ) )  throw UnderflowException( );  return array[ 1 ];  } |

**Deleting (Pulling info) in a MinBH**

* deletion is ***ONLY*** authorized for the MINIMUM value (or index [1])
  + ***not any other value***
* we are not deleting the node, just replace the data inside
* now replaced with the NEXT lowest value
  + which SHOULD be close to the top of the tree
* tree must maintain it’s shape
* but we will delete the LAST complete node in the tree since
  + since now it will be empty

|  |
| --- |
| Deleting a node, and re-heaping |
|  |
| Deleting Min value |
|  |

|  |
| --- |
| Replacing root with LAST value in tree |
| 92 was chosen NOT BECAUSE IT WAS THE MAX!! TOTALLY BY COINCIDENCE. |
| Percolating Down - Comparison |
| Comparing 92 (root) with whatever child is smallest |

|  |
| --- |
| Percolating Down – Match |
|  |
| Percolating Down – Swap |
|  |

|  |
| --- |
| Percolating Down – Comparison |
|  |
| Percolating Down – Match |
|  |

|  |
| --- |
| Percolating Down – Swap |
| Done since no more immediate nodes to compare to. |

Delete the NEXT node using the result above. Answerb:

Delete/Heapify – the function(s)

* deleteMin() and percolateDown()
  + deleteMin is the bootstrap to get things started
  + percolateDown is iterative in comparing and swapping
    - also called **heapify**
    - but recursive, and only goes down one side of the tree
    - takes log n time

|  |
| --- |
| deleteMin() function |
| Java |
| /\*\*  \* Remove the smallest item from the priority queue.  \* **@return** the smallest item, or throw an UnderflowException if empty.  \*/  **public** AnyType deleteMin( )  {  **if**( isEmpty( ) ) { **throw** **new** UnderflowException( ); }  AnyType minItem = findMin( );  array[ 1 ] = array[ currentSize-- ];  percolateDown( 1 );  **return** minItem;  } |
| C++ |
| void deleteMin( )  {  if( isEmpty( ) )  throw UnderflowException( );  array[ 1 ] = array[ currentSize-- ];  percolateDown( 1 );  } |

|  |
| --- |
| percolateDown() function |
| Java |
| /\*\*  \* Internal method to percolate down in the heap.  \* **@param** hole the index at which the percolate begins.  \*/  **private** **void** percolateDown( **int** hole )  {  **int** child;  AnyType tmp = array[ hole ];  **for**( ; hole \* 2 <= currentSize; hole = child )  {  child = hole \* 2;  **if**( child != currentSize &&  array[ child + 1 ].compareTo( array[ child ] ) < 0 )  child++;  **if**( array[ child ].compareTo( tmp ) < 0 )  { array[ hole ] = array[ child ]; }  **else** { **break**; }  }  array[ hole ] = tmp;  } |
| C++ |
| void percolateDown( int hole )  {  int child;  Comparable tmp = array[ hole ];  for( ; hole \* 2 <= currentSize; hole = child )  {  child = hole \* 2;  if( child != currentSize && array[ child + 1 ] < array[ child ] )  child++;  if( array[ child ] < tmp ) { array[ hole ] = array[ child ]; }  else { break; }  }  array[ hole ] = tmp;  } |

Performance

* construction O( n )
  + even if data is out of order, we place in heap with **partial** ordering
  + still stored in a simple array!!
* findMin O( 1 )
* insert O( log n )
* deleteMin O( log n )

Heap Construction – the function

* lays all items into array first, no matter order in construction
  + done in constructor
* then “builds the heap” (sorts, partially) in buildHeap
  + notice that buildHeap uses percolateDown starting at middle of the array
  + this is enough to have the real minimum value “rise” to the top of the heap
* neither of these functions are recursive

|  |
| --- |
| BuildHeap function |
| Java |
| /\*\*  \* Establish heap order property from an arbitrary  \* arrangement of items. Runs in linear time.  \*/  **private** **void** buildHeap( )  {  **for**( **int** i = currentSize / 2; i > 0; i-- )  percolateDown( i );  } |
| C++ |
| void buildHeap( )  {  for( int i = currentSize / 2; i > 0; i-- )  percolateDown( i );  } |

Sorting a Heap – MinBH

* given a list of n values, we can build and sort in O(n log n)
  + insert ***random*** values = O(n)
  + heapify = O(n)
  + repeatedly delete min and re-heapify O(log n) \* n times
* heapify
  + re-ordering the values so Parent is < it’s kids in a MinBH
* delete
  + retrieves the CURRENT minimum node in a MinBH
    - value is saved in another array
  + ***automatically calls percolateDown()***
* this looped OVER and OVER will return a sorted list of items
* this means we will need another array of the same size, just to hold the cast offs
  + UNLESS, we store the values casted back in the deleted code’s position
  + but this will have everything backwards!!

|  |
| --- |
| Using a MinBH to sort |
|  |

|  |
| --- |
| Take 26 (current Min), now heapify |
|  |
| Take 31 (current Min) , now heapify |
|  |

Perform the next two deletions on the heap above. Make sure to draw the tree AND the array

Sorting a Heap – MaxBH

* here we avoid the “backward” issue
* now the parent is >= it’s kids
  + so HIGHEST value is at the top of the heap

|  |
| --- |
| Max Heap Example |
|  |
| Deleting 97 (current Max), now heapify |
|  |

Perform the next two deletions on the heap above

In it’s entirety

<http://nova.umuc.edu/~jarc/idsv/lesson3.html>

MinBH (using arrays) shortcomings

* sorting, wrong order
* merge
  + merging two arrays, no real shortcut
  + so O(n1 + n2)
    - this stinks!!!
  + for an alternative, we have Left-ist Heaps!!!

Answers:

|  |
| --- |
| **Inserting into a Heap Exercises** |
| 1. <http://youtu.be/a9_g-8FclHc> 2. <http://youtu.be/NTMmlXI2CCQ> 3. <http://youtu.be/6X4nRhY2MFc> 4. 3 Swaps |

|  |
| --- |
| Deleting from a Heap |
| <https://www.youtube.com/watch?v=0XPKfBvPcrM&list=PLC7fNkE1QplZ6WB6SC6P0dLgUzhJSPpWr&index=1> |

Sources:

In General

<http://courses.cs.washington.edu/courses/cse332/13wi/lectures/cse332-13wi-lec04-BinMinHeaps-6up.pdf>

Minimum Heap Building/Deletion

<http://www.cs.usfca.edu/~galles/visualization/Heap.html>

Maximum HeapSort

<http://www.cs.usfca.edu/~galles/visualization/HeapSort.html>

Random Heapsort

<http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html>

<http://nova.umuc.edu/~jarc/idsv/lesson3.html>

Code

C++ <http://users.cis.fiu.edu/~weiss/dsaa_c++3/code/BinaryHeap.h>